DRE-PlugIns

For Together Control Center

User Manual

Table of Contents

1 Introduction 3
2 Theory 3
2.1 Dependencies between Use Cases 3
2.2 Dependencies between Use Cases and Classes 4
2.3 Dependencies between Classes 5
2.4 Components 6
3 Installation 8
4 DRE-PlugIns Usage 9
4.1 Generality Levels 9
4.2 Use Cases 9
4.3 Classes 10
4.4 Components 11
4.5 Run DRE-UML 12
4.6 Run DRE-Tool 15

1 Introduction

The DRE-PlugIns for TCC help assessing reuse potential during the design and coding phase of software development. We provide two PlugIns for TCC, namely DRE-UML and DRE-Tool.
DRE-UML enables the user to set generalities for Use Cases and Classes, define relations between Use Cases and Classes, define relations between Classes, define relations between Components and Classes and finally run a measurement on the design to assess reuse potential and provide refactoring guidelines.
DRE-Tool crafts a bridge between TCC and the standalone DRE. It ships over project information, generalities and relations between classes. Using these one can use the standalone DRE to run reusability metrics in code.
This user manual is organized in the following way: section two gives a short introduction to the underlying theory, section three contains installation instructions and section four finally shows how to use the DRE-PlugIns.

2 Theory

This chapter is only supposed to give a quick and short overview of the underlying theory. For more detailed information check out the following paper on our website:

F. Eickhoff, J. Ellis, S. Demurjian, D. Needham: A Reuse Definition, Assessment, and Analysis Framework for UML

To assess the reuse potential of a design we take measurements on four different levels:

- Dependencies between Use Cases
- Dependencies between Use Cases and Classes
- Dependencies between Classes
- Dependencies between Components and Classes

For all dependencies we defined a set of properties and according refactoring guidelines. The properties listed in the remainder of this section are borrowed from our paper.

2.1 Dependencies between Use Cases

Each Use Case gets assigned a level of generality G. Dependencies between Use Case are established by <include>, <extend>, <generalize>. Therefore we get the following properties:

Property 4. UC_A extends UC_B is a relation in which UC_A adds behavior to UC_B, meaning that UC_A is at most as general as UC_B or $G_{UC\text{-}B} ? G_{UC\text{-}A}$.

Property 5. UC_A includes UC_B is a relation of the behavior sequence of the supplier UC_B into the interaction sequence UC_A, meaning that UC_A is at most as general as UC_B or $G_{UC\text{-}B} ? G_{UC\text{-}A}$.

Property 6. UC_A generalizes UC_B is a relation of a specialized UC_B to a more general UC_A, meaning that UC_B is at most as general UC_A or $G_{UC\text{-}A} ? G_{UC\text{-}B}$.

The refactoring guidelines:

RG1, RG2: To enforce Property 4 or Property 5, the refactoring rule is: If $G_{UC\text{-}B} > G_{UC\text{-}A}$ then refactor by making UC_B *more general* so that $G_{UC\text{-}B}$? $G_{UC\text{-}A}$ or UC_A *more specific* so that $G_{UC\text{-}B}$? $G_{UC\text{-}A}$, or by *removing the extend/include.*

RG3: To enforce Property 6, the refactoring rule is: If $G_{UC\text{-}A} > G_{UC\text{-}B}$ then refactor by making UC_A *more general* so that $G_{UC\text{-}A}$? $G_{UC\text{-}B}$ or UC_B *more specific* so that $G_{UC\text{-}A}$? $G_{UC\text{-}B}$, or by *removing the generality*.

2.2 Dependencies between Use Cases and Classes

Both Use Cases and Classes have generality levels G. Use Cases are related to a set of classes C_A that realize the Use Case. Also Classes are related to other classes. We get the following properties and refactoring guidelines:

Property 7. Suppose that UC_A is related to a set of classes $C_A = \{C_1, C_2, \ldots, C_n\}$ for some *n*. Then, the generality of UC_A must be as specific as the most specific class in C_A, and may be more specific, i.e., $G_{UC\text{-}A} = \max\{\text{generality ? } C_i \text{ ? } C_A\}$.

RG4: To enforce Property 7, the refactoring rule is: generality change of UC_A or one or more C_i ? C_A until $G_{UC\text{-}A} = \max\{\text{generality ? } C_i \text{ ? } C_A\}$, or the removal of all classes in C_A that cause the $G_{UC\text{-}A} = \max\{\text{generality ? } C_i \text{ ? } C_A\}$ to be violated.

Property 8. Suppose that UC_A is related to C_A and UC_B is related to C_B. If UC_A is related to UC_B (extend, include, or generalize – see Section 3.1), then there has to be at least one transitive relation chain from one C_i ? C_A to one C_j ? C_B.

RG5: To enforce Property 8, the refactoring rule is: add one or more dependencies between class(es) C_i ? C_A and class(es) C_j ? C_B.

Property 9. Suppose that UC_A is related to C_A and UC_B is related to C_B, and that UC_A is not related to UC_B, that is, there is *not an* extend, include, or generalize relation. Then, if at least one C_i ? C_A is related directly (meaning not by transitive relation chain) to at least one C_j ? C_B, there must be a refactoring to correct the problem.

RG6: To enforce Property 9, the refactoring rule is: make UC_A related to UC_B or remove all dependencies between all C_i ? C_A and all C_j ? C_B.

Property 10. Suppose that UC_A is related to C_A. Then for all C_i ? C_A, if there is a transitive relation chain from C_i to some C_j ? C_A, then UC_A must also be related to C_j in some manner. UC_A can be either related directly to C_j or it can be related to some UC_B, to which C_j is related

RG7: To enforce Property 10, the refactoring rule is: ? C_i ? C_A related to some C_j ? C_A include C_j in C_A (C_A= C_A ? C_j), or relate UC_A to some UC_B where C_j ? C_B, or unrelate UC_A to C_i (C_A= C_A - C_j).

2.3 Dependencies between Classes

Again classes get assigned a level of generality. Also we want to relate classes to other classes, meaning that these classes are intended to work together. On the other hand we get couplings between classes by UML associations and dependencies, attributes, method signatures, method invocations and later on in the source code.
We define the following properties:

Property 1. In an inheritance hierarchy, the parent class is equally general or more general than its children, meaning that children have at least the reusability level of the parent, plus optional specific (less general) method(s). For example, WalmartItem is less general than its ancestor Item.

Property 2. The generality level of a class is equal to the generality level of the least reusable coupled class. Given two classes, C_p and C_q with generality levels G_p and G_q there are two cases: C_p is more specific than C_q ($G_p > G_q$), in which case C_p is the least reusable class; and C_p is more general than C_q ($G_p < G_q$). If there is a coupling from C_p to C_q, C_p may be dependent on specific features of C_q that will condition the reusability of C_p to situations when C_q is reusable, i.e., the generality level of C_p must be equal to or greater than the reusability level of C_q.

Property 3. Couplings between unrelated classes are undesirable and hinder reuse, since unrelated classes are not intended to be reused together in future applications. If there are dependencies among unrelated classes, the coupled classes must be brought along to the reusing application in order for the system to function properly, which contradicts the intent of not reusing the components together.

We also the following coupling types, where G means “General” and S means “Specific” (if we have more than two levels, we only compare two generalities, so they can be seen as just general and specific):

Type 1. G ? G among related classes is an asset to reuse, the two classes are to be reused together, and the objective is to increase these couplings.

Type 2. G ? G among unrelated classes is undesirable since the source and destination are not expected to be reused together. Refactor by moving both the source and destination to Specific descendant classes or making classes related.

Type 3. G ? S among related classes is undesirable, since the General class (to be reused) depends on a class which is not expected to be reused. Refactor by moving the source to a Specific descendant or destination to a General ancestor.

Type 4. G ? S among unrelated classes is undesirable since the source is expected to be reused while the destination is not. Refactor by moving the source to a Specific descendant class.

Type 5. S ? G among related classes does not hinder reuse since the source of the coupling is not expected to be reused. Refactor to improve reuse by moving the source to a General ancestor.

Type 6. S ? G among unrelated classes does not hinder reuse since the source of the coupling is not expected to be reused. There is no need to refactor in this case.

Type 7. S ? S among related classes does not hinder reuse since the source of coupling is not expected to be reused. Refactor to improve reuse if both the source and destination are moved to their General ancestors.

Type 8. S ? S among unrelated classes represents the desired situation for couplings between unrelated classes; they need to be among the Specific classes.

2.4 Components

Component Diagrams offer another opportunity to assess reuse potential. Using UML-Component diagrams the developer can model components, the interfaces they implement and the dependencies between components.
In general UML components don't have to contain classes or code. They can just represent physical units of a system. But for our approach we consider components as a collaboration of classes, which together provide a coherent functionality.

Definition 4:

A Component is related to a set of Classes C_C, which implement the Components functionality and behavior.

As components are self-contained physical units, it should be easy to replace a component with another. To achieve that we need low couplings between components. Therefore components should only be accessed through the interfaces they implement, or through classes that act as facades for the component.

Definition 5:

Each Component gets assigned a set of Classes ?, which act as facades for the component and hide the implementation details of the component.

As the UML does not define relations between Class-interfaces (meaning Interfaces/Abstract Classes in code) and Component-Interfaces we need to define that.

Definition 6:

Each Component-Interfaces gets assigned a set of Class-Interfaces ?, which are the Class-Interfaces/Abstract Classes, that the Component-Interface represents.

As Components are supposed to be easily replaceable and plug able, it makes no real sense to assign generalities to Components, because the definition of a Component itself makes it general in any case.

Based on the Definitions shown above, we get the following properties:

Property 11: A component is related to a set of Classes C_C. Classes in C_C have relations between them. The relations between the classes form one or more graphs. As we want a component to be cohesive there should only be one relation graph.

RG8: To refactor property 1, the software engineer should either establish relations between the graphs, or split the component into several components, which don't violate Property 1 any more.

Property 12: A class C_C in C_C is related or coupled to a class C_O of another component. If C_O is not a Façade-Class or a Class-Interface of the Interface that the other component implements there has to be a refactoring.

RG9: The software engineer either has to remove the relation/coupling form C_C to C_O or make the C_O a Façade-Class or Class-Interface of an implemented Interface.

Property 13: A class C_C in C_C is related or coupled to a Façade-Class or a Class-Interface of another component, but there is no dependency link to the other component or the interface it implements.

RG10: Either remove the relation/coupling or introduce a dependency link to the other component or one of its interfaces.

3 Installation

As a prerequisite for the DRE-PlugIns you need an installed TCC 5.5 or higher (www.togethersoft.com).
If you want to use DRE-Tool-PlugIn you also need an installed DRE, which is available at our website.

The installation of the DRE-PlugIns is pretty simple. If not done yet, download the DRE-PlugIn ZIP form our webpage (http://www.engr.uconn.edu/~steve/DRE/tcc.html).
Unzip DRE_TCC_Plugin.zip into your TCC installation folder. Make sure to use the folder information of the ZIP, so that the files get into the right directories.

For example:
You installed TCC to C:\Program Files\Together.
Unzip DRE_TCC_Plugin.zip to C:\Program Files\Together.

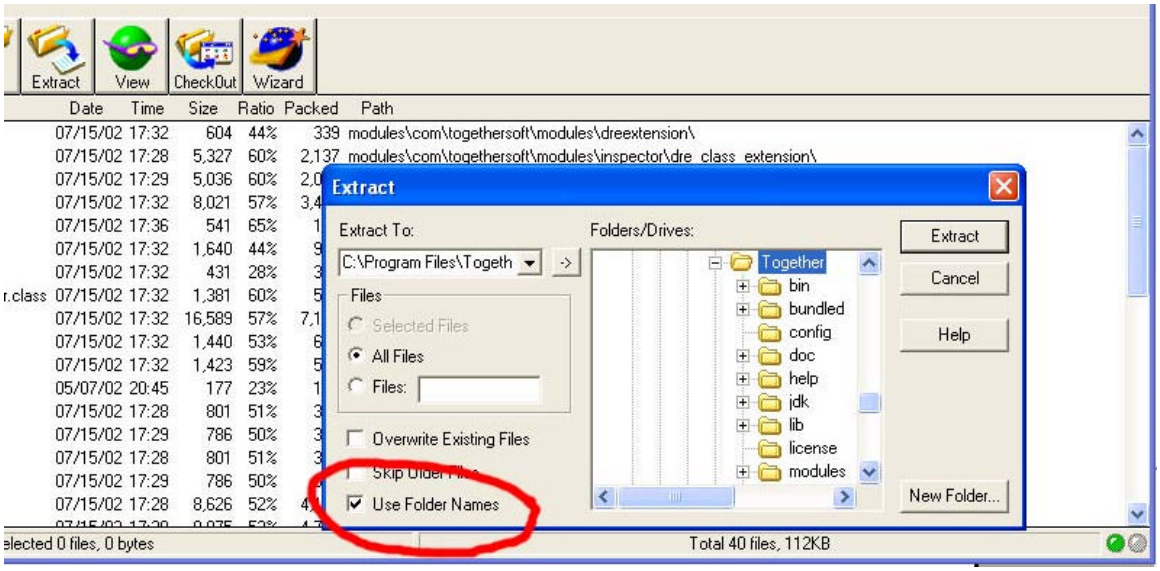

Figure 1: Make sure to use Folder Names

After doing that the DRE-UML-PlugIn should run without problems.

To get the DRE-Tool-PlugIn running you have to modify a configuration file to match your installation path of the standalone DRE.
Open ..\Together\modules\com\togethersoft\modules\dretool\Manifest.mf in an arbitrary editor. What you'll see should look like this:

```
Name: DRE-Tool
Time: User
Main-Class: com.togethersoft.modules.dretool.newDRE
Class-Path: $TGH$\com\togethersoft\modules\dretool;c:\DRE
Folder: "DRE-PlugIns"
```

You have to change `Class-Path` so that TCC can find the standalone DRE. In this case it is installed in c:\DRE.

4 DRE-PlugIns Usage

This section shows you how to use the DRE-PlugIns in TCC. According to the underlying theory you have to set several DRE-properties for Use Cases and Classes.

4.1 Generality Levels

The DRE-PlugIns support different numbers of generality levels for each project. The default number is four. If you need more or less generality levels for your project you can change this value. Open the Project options dialog of TCC (Menu: Options – Project Options…).
The following dialog will show up:

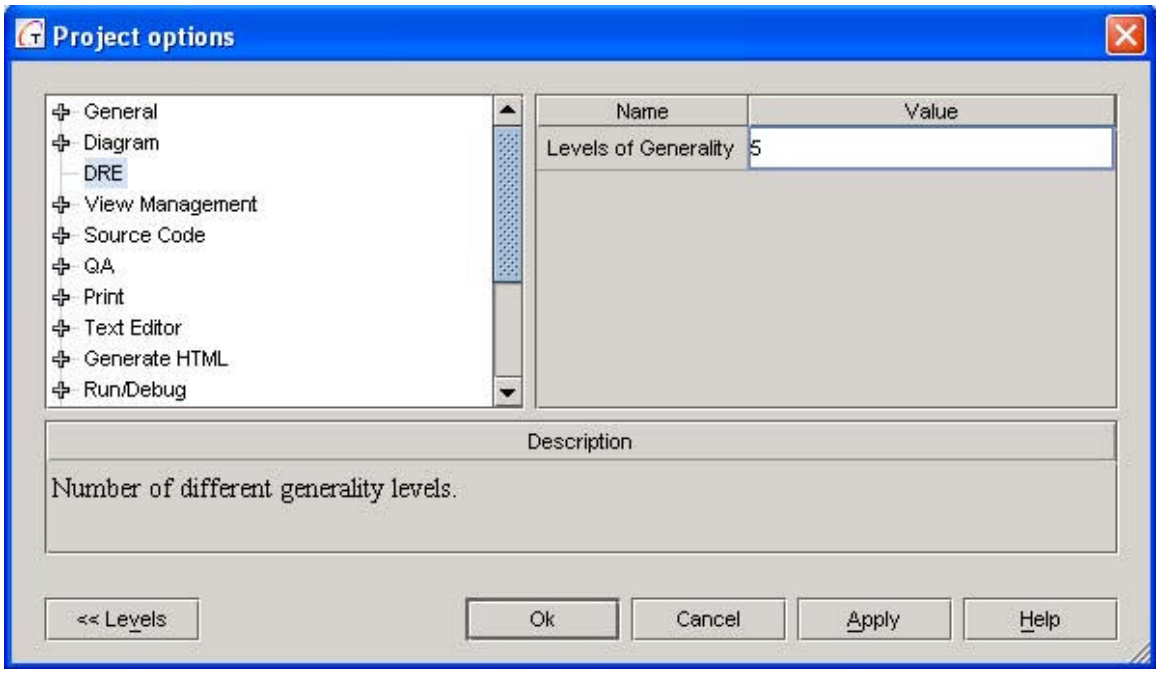

Figure 2: Set number of generality levels

Select the DRE-Folder and enter the desired value.

4.2 Use Cases

For each Use Case you have to set the level of generality. To do that, open the Inspector-Dialog for the Use Case and select the DRE Page. Using the Drop-down Box you can set the generality:

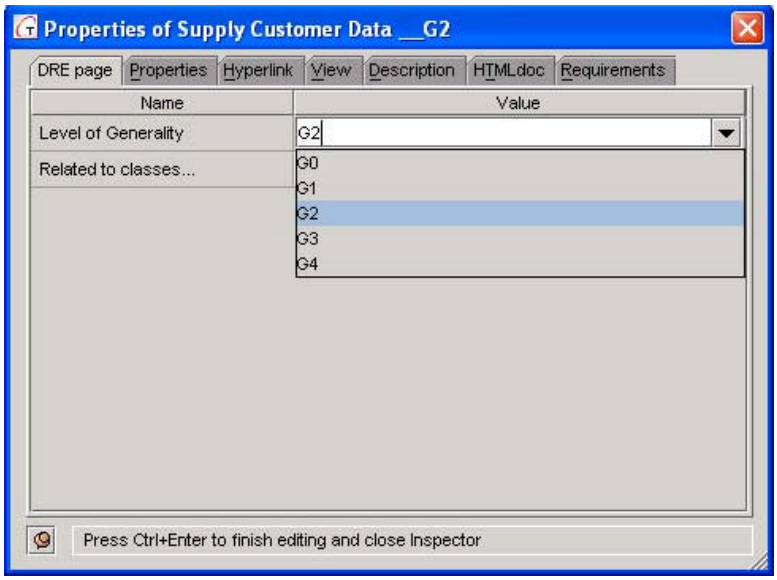

Figure 3:Setting the generality for a Use Case

Also you have to relate Classes to any Use Cases that they will realize. To do that, open the Inspector-Dialog of the Use Case and select the DRE Page. Press the button right of the "Related to Classes" edit field to relate classes to this Use Case:

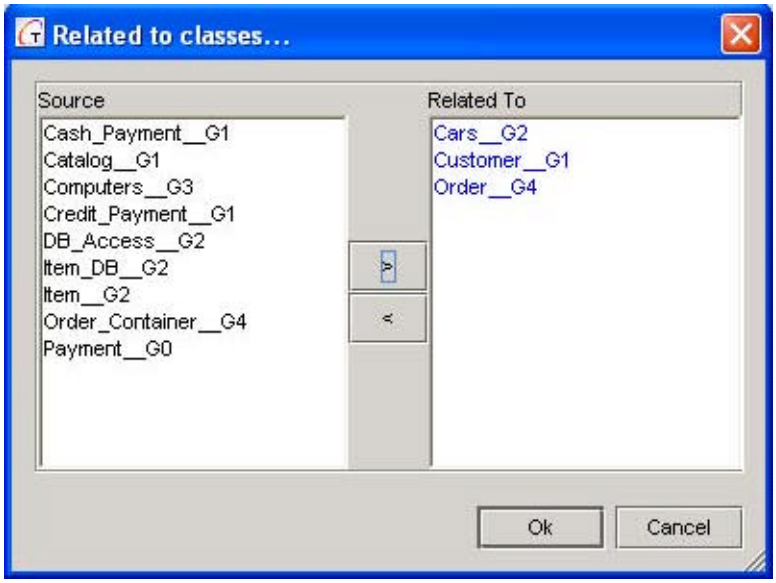

Figure 4: Relate a Use Case to Classes

4.3 Classes

Setting the DRE-properties for Classes works basically the same way as for Use Cases. Open the Inspector-Dialog for a Class and select the DRE Page. There you can set the generality, the related classes and the Use Cases that use this class.

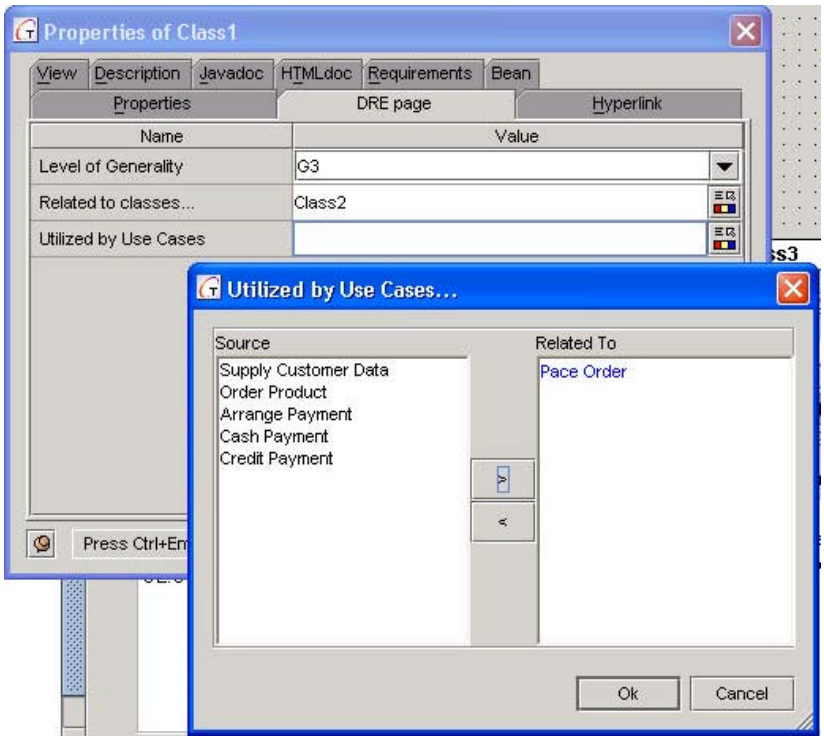

Figure 5: Class Inspector and Setting relations to Use Cases

4.4 Components

To set the DRE-properties for Components you also have to use the Inspector dialog's DRE-Page. There you can set the Classes that are used by the component and its façade classes. Setting the properties for the Component-Interfaces works the same way. Open the Inspector for the Component-Interface and select the DRE-Page. There you can set the implemented interfaces.

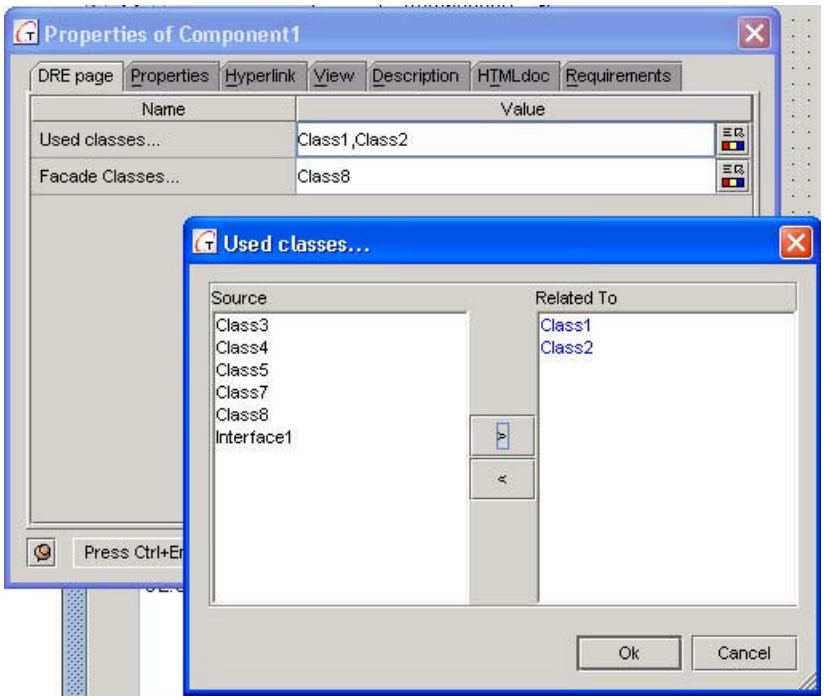

Figure 6: Component Inspector and relating Used Classes

4.5 Run DRE-UML

Once you have set all necessary DRE-properties for Use Cases, Classes and Components you can run the reuse assessment by selecting the TCC Modules Tab, open the DREPugIns folder, right click DRE-UML and select run:

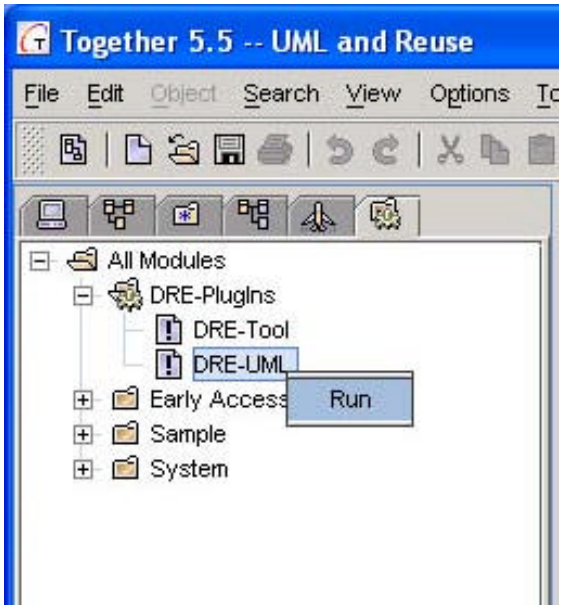

Figure 7: Run the DRE-UML

The results are shown in the DRE Warnings pane:

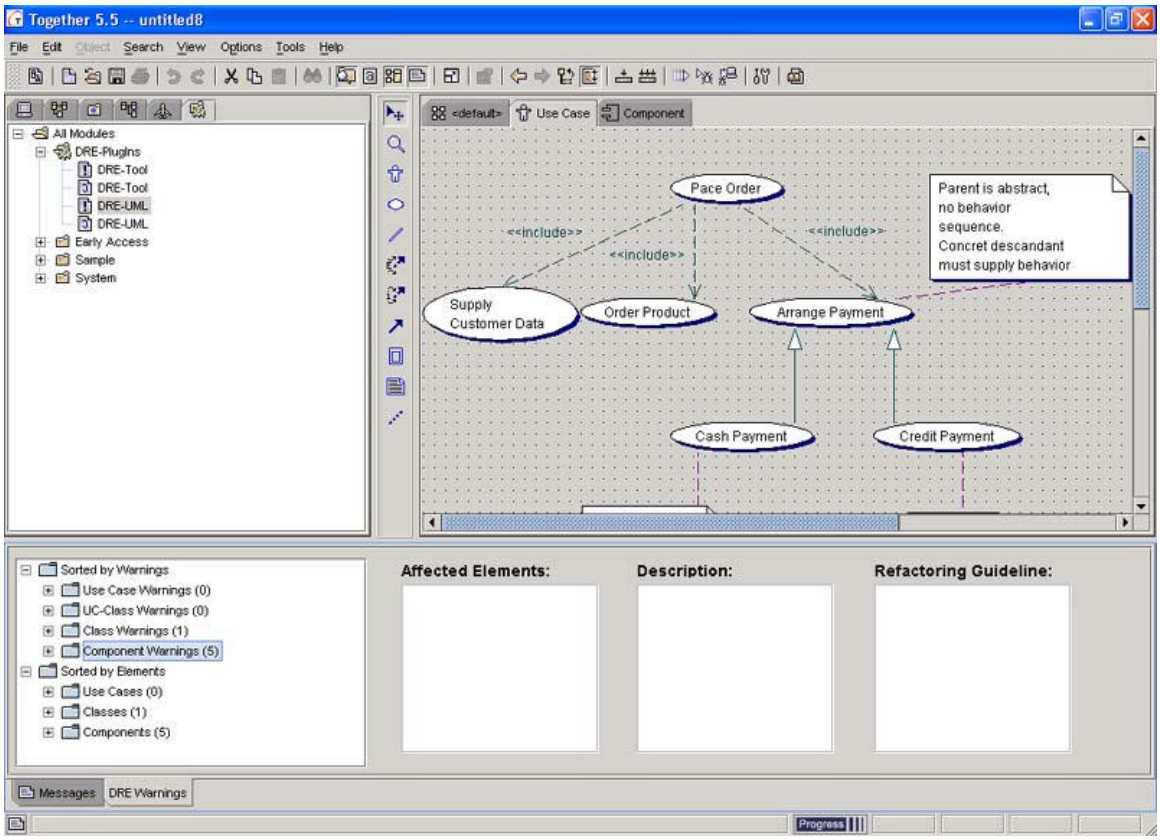

Figure 8: DRE-Warnings

The Warnings Pane is organized in the following way:
On the left side you see a tree with all warnings. Basically there are two trees, one sorted by Warning-type and one sorted by Elements. You can use the first one if you want to track down warnings of a specific kind. The second one should be used to track down warnings for a certain Use Case, Class or Component.
After selecting a warning the boxes on the right side show more information about it:

- **Affected Elements**: Shows a list of all elements that are affected by this warning. If you click on one of those the corresponding diagram will be shown automatically with the element selected (this works only if the diagram is already open)
- **Description**: Provides a description of the warning
- **Refactoring Guideline**: Provides one or more refactoring guidelines for the currently selected warning

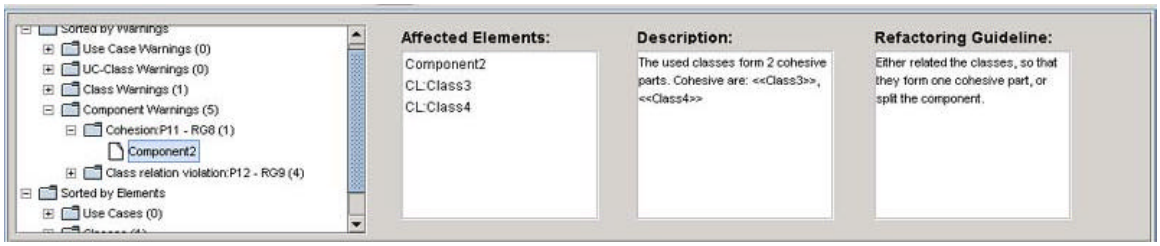

Figure 9: DRE-Warnings

As you can see in the figure the nodes in the tree provide a link to the properties and refactoring guidelines of the underlying theory (e.g. P11-RG8). This additional information may help you improve the reuse potential of your design.

The DRE-UML runs it's warning based on the following properties:

- Use Case Diagrams:
  - Generalities
  - Dependencies to other Use Cases (include, extend, generalize)
  - Related Classes
- Class Diagrams:
  - Generalities
  - Dependencies to other Classes (association, dependency, generalization)
  - Related Classes
- Sequence/Collaboration Diagrams:
  - Method invocations
- Component Diagrams:
  - Related Classes
  - Façade Classes
  - Implemented Interfaces
  - Dependencies between Components and Interfaces

4.6 Run DRE-Tool

In order to run the DRE-Tool you have to have generalities of classes and relations between classes set. As the DRE-Tool works on source code you should have at least some code. It is not necessary to mark the Use Cases, if you only want to use the DRE-Tool.

To run the DRE-Tool select the TCC Modules Tab, open the DREPugIns folder, right click DRE-Tool and select run.

The standalone DRE-Tool shows up and you can run measurements on your code. Changes of generalities or relationships between classes are shipped back to TCC as soon as you close the DRE-Tool.

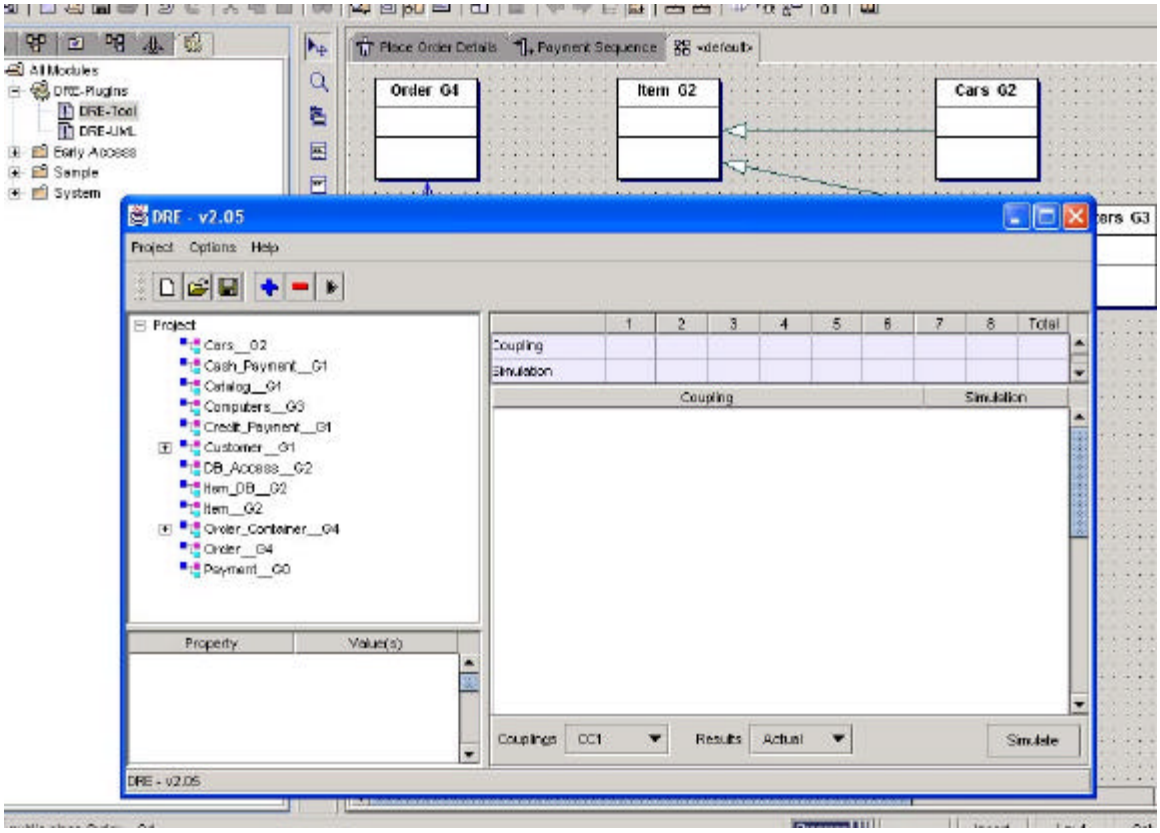

Figure 10: DRE-Tool started from TCC